Miles:

Thank you for Fortran. It has become more civilized both in its own wording and in stylistic rules it imposes upon me as a programmer, and has acquired a lot of attractive new capabilities (modules, first of all), so I am crossing fingers for its survival.

After a first reading, some remarks.

The main problem is to be able to use Fortran in a less conventional environment. Not much is needed.

There should be some sort of "ON condition", quite uncomplicated, something like this. An "event" (identified by a name or a numeral) is either predefined by the processor (a clocking or an error condition or some external action or whatever) or has no inherent meaning, its occurrence being simply the execution of a program statement, say, GO(event). A consequence of an event can be defined by a statement ON (event) clt. subroutine; this is redefinable, and revocable by NO (event). If at event occurrence a consequence is defined, the subroutine gets called, otherwise it has no consequence. Predefined events have predefined (but redefinable) consequences. This is incredibly simple to implement. — Some "emergency exit" should be added that (rather than STOPping) orderly returns to a label contained and defined in the main program.

There should be, if not a semaphore, at least some operation that reads-and-changes a variable as a single indivisible action.

File handling is tailored very much to single-customer needs. I cannot stipulate any protective measures in a standardized way, and, e.g.,

whenever I am permitted to open a file, I am also permitted to delete it (9.3.5.1). What is worse, there are no provisions for file sharing, multiple access. This in fact could be solved by obvious extensions of the module and interface concepts. — Some polishing is needed in other respects as well. Nothing is said about what happens, e.g., in case of a name collision with STATUS = 'NEW' (9.3.4.2). Record length, even if processor-dependent, ought to be computable from within the program, using standard inquiries.

Curiosity should be better satisfied: e.g. size of a file (number of records), its record length etc.; available memory (to be used, e.g., with allocations). Perhaps even some environmental inquiries could be standardized (is there a console on line? a printer available?). (Information is generally scarce. Thus I am not supposed to inquire about rank. Why?)

Typographical capabilities of present-day consoles, printers etc. are far superior to anything that can be described in Fortran. What I want is not a full graphics processor within Fortran but a non-disgusting appearance, better positioning possibilities (at least to be able to program a questionnaire) and the like. That much <u>can</u> be standardized. Again, inquiries are needed (as to sizes etc.). — By the way: With list-directed and namelist output (10.8.2, 10.9.2.1), I would prefer the processor-dependent w, d, e to be merely default values, that is, redefinable (as it is permitted with DELIM etc.). With format-incompatible numerical output, instead of asterisks (10.5.1 (5)), I would like the layout destroyed, to have at least <u>some</u> information. It would be sometimes convenient to read a record piecemeal (like output records can

be put together by starting continuation records with "+"). This is handy, e.g., in implementing self-defining input objects (which of course can be done over the byway of internal files, yes) or non-record input (which cannot).

Some simple macro capability ought to be added as part of the standard. This amounts to "defining non-standard features in a standard way". The problem of implementing "national standards" would be essentially solved, and everyone could have his personal lingo and shorthand. Moreover, it would enable the programming of "infix routines", and that in a portable way.

A lot of minor amendments could be made at practically no cost.

We have the "models" for integer and real data (13.6.1). We could define, as a type, a standard representation of their binary version, and provide intrinsic functions to convert to/from it; of course, "default" and "model" may happen to co-incide, and conversion to be the identity. (All this would be analogous to IACHAR etc.)

Logicals are heavily relied upon. So I might like to specify "non-default" logicals (as I can specify non-default reals). To be frank: one-bit logicals. This offers a trade-off between time and space. — Maybe the same for integers; but this is less important.

Why not have bit operations? Instead of extra type-complications, they could be performed on logicals and on "model" types (or on intrinsic types, meaning implicitly the result as defined by the "model"

operation – which would be an immediate operation in most implementations).

Instead of (or along with) PRESENT, initial/default values for optionals?

Pointers would be convenient, even to procedures.

Alphanumerical labels (terminated with ":") could be permitted, at least in free form (they improve upon program legibility). EXIT and CYCLE targets (8.1.4.1) are a timid step in a similar direction. Why not go the whole hog?

I don't think that computed go to should be going to be obsolete (B.3.2). True, it can be replaced by an equivalent "case". Still, it can be implemented much more efficiently, arises quite naturally in many contexts, and requires less clerical work to write down. Discard, rather, such freaks as MAX1 and AMAXØ or the fossil blank-is-zero. (Incidentally, Deprecated Features have no Nature: there is no 1.6.3. Maybe they do not become obsolete anyway.)

Now senseless prohibitions. I write, say, a matrix multiplication subroutine MATMULT(Z,X,Y) and want it to run with X,X,X. (I sure can write it to work properly.) Well, this is forbidden (by 12.5.2.9). – Unfortunately, the standard has retained the bad habit of intermixing, on an equal-rights basis, syntactic rules and functional descriptions with obscure semantic threats and esoteric programming advices. What I am interested in is, not what I am supposed to do, but what is going to happen. In this particular case, a little definitory effort could have made the above call legal in any implementation, even without specifying X,Y as IN and Z as OUT. – And, in fact, I want to have a say, quite

generally, as to how my operands are accessed, through pointers, in save areas, over the stack, on stone tables, and especially whether or not huge arrays get repeatedly copied. Fortran leaves this in many cases to the discretion and valour of the implementor, who has to choose uniform methods; I can only guess and hope for the best.

Presumably unintended: As MOD is defined (13.12.66), there are three residue classes mod 2, viz. 0, 1, −1. (Mathematics are not the strong point of the Committee. See, e.g., the not diehardly conservative interpretation of the term "Fourier series" in C.13.2.1-2.) — I would like to have an intrinsic function for entier in the mathematical tradition (where ENTIER (−0.5) equals −1), or better two functions: floor and ceiling (floor(x) is entier(x), ceiling(x) is −entier(−x)).

The draft is uneven, both in the definiens and in the definiendum. As to the second, one would wish for more "orthogonality", more analogous treatment where there are analogies. Thus, e.g., redundant statements are legal in some cases, illegal in others. This cannot be ironed out now. Perhaps in Fortran 99. Text style and formulations can be improved upon, however. Large parts are well written and clear. But many a passage is unkempt. A.1.3 states that deleted features are candidates for removal from future versions of the Standard. The closing sentences in 13.4.4, 13.6.2 and 13.7.5 make no perceivable sense, if not intimating that those functions are "optionally standard", which does not make sense either. Many passages are understandable, but understanding has been made difficult by the wording. The draft abounds in pedantic verbosity. Instead of stating, once for all, that everything that is referenced must have been defined, ~~this is~~ separately formulated for every instance, not

only in different places, to help the forgetful, but five times in a row within a single short passage (7.1.7, in the beginning). You read the instances one by one, looking for some hidden subtlety that accounts for the separate wording. There is none. Or: The draft states (8.1.4.4.4(6)) that a DO loop is terminated ~~if~~ when "a STOP statement anywhere in the program is executed, or execution is terminated for any other reason". If this is really needed here, it is needed in the description of all the other constructs, units etc. as well. But it is missing there. So put it in. Similarly, I strongly suggest that the interpretation of "must" and "must not" (1.5) be supplemented with interpretations of "may" and "may not" as well as "is" and "is not". Moreover, an interpretation of "is to be", that is used in interpreting "must" and "must not", should be added.

Indeed, the whole thing could be condensed to ~~30–35~~ 35–40 % of its present volume and be easier to understand.

Use what you can, and don't feel obliged to produce an answer, official or unofficial.

A viszontlátásra,

Bródy